

DAA/LANGLEY
NAG-511

1N-61

64831 CR
P.93

Annual Progress Report
Grant No. NAG-1-511
September 1, 1984 - January 31, 1988

SECOND GENERATION EXPERIMENTS IN
FAULT TOLERANT SOFTWARE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Dr. Dave E. Eckhardt, Jr.
ISD M/S 478

Submitted by:

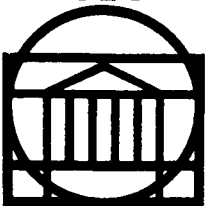
J. C. Knight
Associate Professor

(NASA-CR-180675) SECOND GENERATION
EXPERIMENTS IN FAULT TOLERANT SOFTWARE
Annual Progress Report, 1 Sep. 1984 - 31
Jan. 1988 (Virginia Univ.) 93 p Avail:
NTIS HC A05/NF A01

N87-27419

Unclas
CSCL 09B 63/61 0064831

Report No. UVA/528235/CS87/104
March 1987



SCHOOL OF ENGINEERING AND
APPLIED SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

83127208

UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA 22901

Annual Progress Report
Grant No. NAG-1-511
September 1, 1984 - January 1, 1988

SECOND GENERATION EXPERIMENTS IN
FAULT TOLERANT SOFTWARE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, Virginia 23665

Attention: Dr. Dave E. Eckhardt, Jr.
ISD M/S 478

Submitted by:

Dr. J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528235/CS87/104
March 1987

Copy No. _____

SECTION I

INTRODUCTION

The purpose of the work performed under this grant is to begin to obtain information about the efficacy of fault-tolerant software by conducting two large-scale controlled experiments. In the first, an empirical study of multi-version software is being conducted. This experiment will be referred to as the "MVS" experiment in this report. The second experiment is an empirical evaluation of self testing as a method of error detection and will be referred to as the "STED" experiment.

The MVS experiment is being conducted jointly by NASA, four universities, and Charles River Analytics, Inc. The participating universities are North Carolina State University, the University of California at Los Angeles, the University of Illinois at Urbana-Champaign and the University of Virginia. During the current grant reporting period, the work at the University of Virginia in the MVS experiment has centered around the preparation of an environment for testing the subject programs and the performance of a set of preliminary tests. Other elements of the experiment are being carried out at the other sites.

The purpose of the MVS experiment is to obtain empirical measurements of the performance of multi-version systems. Twenty versions of a program have been prepared at four different sites (the universities) under reasonably realistic development conditions from the same specifications. The experimenters are now preparing to evaluate these programs in various ways, in particular by extensive dynamic testing.

The STED experiment is being conducted jointly by the University of Virginia and the University of California, Irvine. The purpose of the STED experiment is to obtain empirical

measurements of the performance of assertions in error detection. Eight versions of a program have been modified to include assertions at two different universities under controlled conditions.

During the grant reporting period, the experiment has been designed, the various programs enhanced with self checks, and the resulting error detection performance measured. The preliminary results of the experiment have been written up as a conference paper.

In this report, we describe the overall structure of the testing environment for the MVS experiment and its status in section II. In section III, we describe a preliminary version of the control system that has been implemented for the MVS experiment to allow the experimenter to have control over the details of the testing. We summarize our work in the STED experiment in section IV. The paper describing the STED experiment is included as Appendix A. In Appendix B, we present the results of the preliminary testing of the programs generated in the MVS experiment.

SECTION II

MULTI-VERSION SOFTWARE TEST ENVIRONMENT STRUCTURE

The basic layout of the test environment was decided at joint meeting of the research members held in Boston in April, 1986. This basic layout has been extended in various ways as a result of numerous discussion and changes in requirements. A fundamental goal of the environment is to be as independent of the machine used for the testing as possible. Thus, although built and distributed as a UNIX based system, the environment should run on other machines with little change.

The philosophy of the test system is to allow the experimenter to specify the initial conditions and sensor failure requirements for a single simulated flight and then to generate a series of acceleration values that are supplied to the programs along with the initial and failure conditions. This is intended to simulate a single flight of an aircraft.

The system allows the experimenter to specify that several (perhaps many) flights are required each with a different (but perhaps similar or related) set of initial conditions. For example, some parameter might have to be varied systematically over some range. In this case, the system will create a sequence of initial conditions in which the required parameter is varied but the same set of acceleration and Euler angles is used for each simulated flight in the set. The systematic variation and the reexecution of the programs on the set of accelerations is handled by the execution environment.

The environment consists of a set of programs that are organized into five tiers or levels. The general form is shown in figure 1. The interfaces between the levels are precisely defined. Each consists of character files thereby permitting the greatest degree of machine independence. The details of the interfaces are referred to here as formats. For example, format 0 describes the

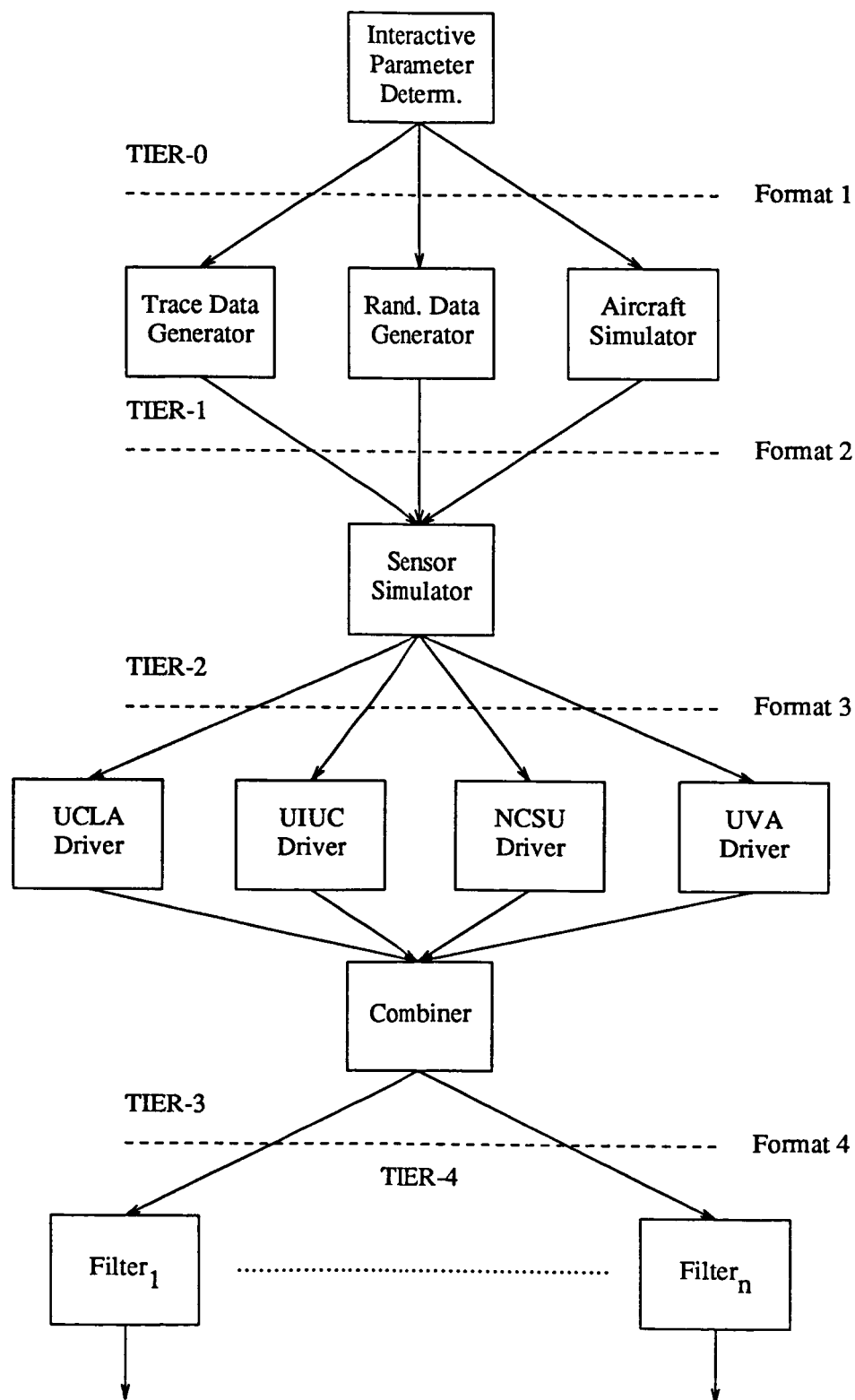


Fig. 1 - Overall Environment Layout

interface between tier-0 and tier-1, and consists of a single explicitly named file. Other formats use more than one file, including in most case the standard input and standard output files. Figures 2, 3, 4 and 5 show the input and output details for each tier individually. In these figures, a dashed line represents either standard in or standard out, and a solid line connected to a named ellipse indicates an explicit disk file. The exact content of the formats is described in the section III.

Tier-0

The purpose of the single program in tier-0 is to interact with the experimenter to determine the parameters of the tests that have to be run. This program produces a file of data for control of subsequent programs after gathering the details of the required tests from the experimenter. The program makes no decisions and generates no data itself (except defaults) so the output datafile contains everything that the experimenter supplied. For simple tests, most parameters can take the default values allowing the definition of the tests to be created with very little input from the experimenter.

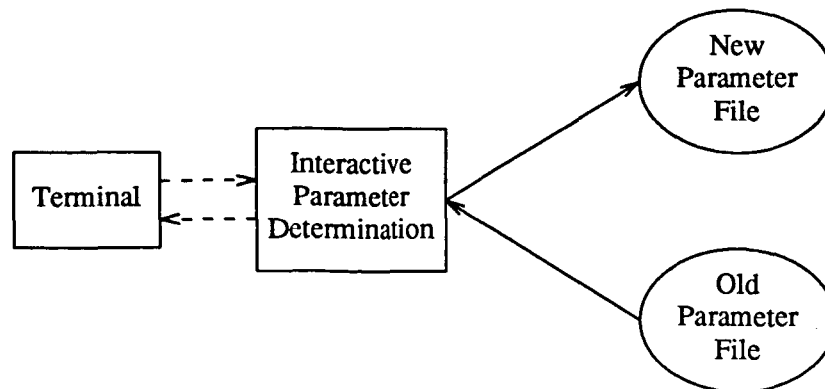


Fig. 2 - Input And Output For Tier-0

Optionally, an existing parameter file can be read by the program to provide a set of initial conditions for the interaction with the experimenter. Thus if two sets of tests are to be run with minimal change between runs, the data file from one can be read in to set values initially for the experimenter during the interaction. A second data file that differs little from the first can be generated merely by indicating the changes.

As will be seen from the discussion in section III, the interaction carried out by the tier-0 program could result in the specification of a large number of tests. The number is computed and supplied to the experimenter for confirmation.

The tier-0 program is written in Pascal. Although it is interactive, the program operates in a very simple menu style to ensure independence of terminal characteristics,

Tier-1

The programs in tier-1 obtain the specifications for the initial conditions from the file that the tier-0 program creates. They then generate the series of accelerations and angles that is required for the specified test flight(s).

There are three programs in tier-1. Each operates with the same input and output interfaces (formats 0 and 1), and as far as the rest of the environment is concerned, they are equivalent. The first generates a series of accelerations from a trace file. It merely reads accelerations obtained from measurement on the B737 aircraft and converts them to the format required by the following tier. This program is written in Pascal.

The second program generates accelerations and angles randomly. Although these values are unrealistic, they are adequate for testing. This program is written in Pascal.

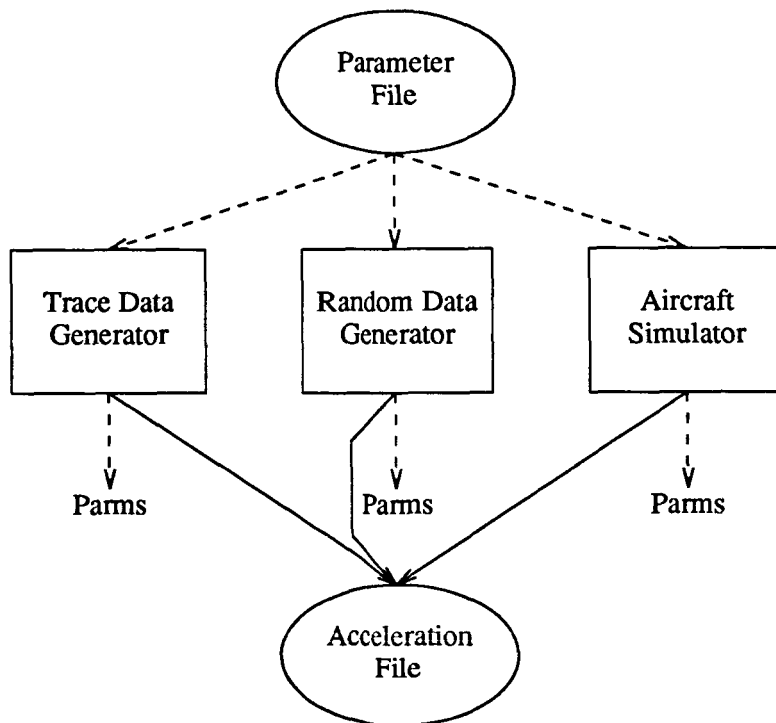


Fig. 3 - Input And Output For Tier-1

The third program is an aircraft simulator that generates realistic values for the required data. This program is being prepared by Charles River Analytics. It is written in FORTRAN.

Tier-2

There is a single program in tier-2, the sensor simulator. This program is written in FORTRAN and the original version was supplied by Charles River Analytics. This program has been modified by the University of Virginia to include the necessary loops for driving the following tiers where parameters are being varied in a series of simulated flights. The program takes an acceleration value and other parameters supplied from the data file generated by the tier-0 program and generates the corresponding sensor values.

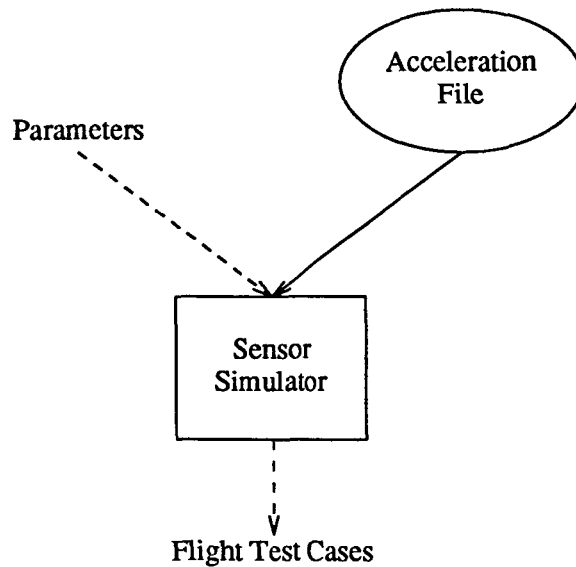


Fig. 4 - Input And Output For Tier-2

Tier-3

Tier-3 contains the actual versions under test, several driver programs, and a utility program. It was deemed inadvisable to attempt to run all 20 versions together with a single driver. Merely compiling a major program of that size would take considerable resources. We have found that several Pascal compilers available to us were unable to compile such a program.

To avoid these problems, tier-3 contains four drivers, one for each university. They read the same inputs but create their own output files and so can be run in parallel. A utility program (the combiner) is then used to combine the output files so that they appear to have come from a driver that executed all twenty programs together. The combiner merely reorganizes the output files of the four drivers. It makes no content changes to the data.

Flight Test Cases

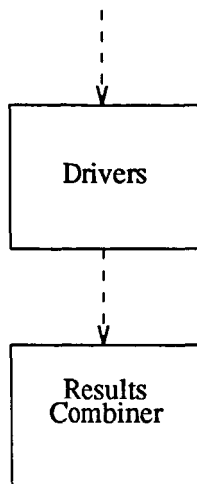


Fig. 5 - Input And Output For Tier-3

The tier-3 drivers keep the initial conditions for a particular flight as global data while executing the required versions. This data includes the calibration data. Thus although each program thinks that it will operate on a single flight acceleration value, the calibration and other parametric information is identical for each acceleration for a flight and so the effect is to have the program do calibration followed by a series of acceleration values.

Tier-4

Tier-4 consists of an arbitrary number of "filter" programs that read the output of tier-3 and do useful things with the output. As new functions are required, new filters will be added. Present filters include programs to allow formatted printing of the raw test results in various forms. Filters are being developed to allow the raw data produced by the tests to be stored in as compact a form as possible on tape. The purpose of these filters is to allow tests to be run and their entire output to be saved for later analysis. This will permit tests to be performed without

all possible analyses being performed at the same time.

SECTION III

MULTI-VERSION SOFTWARE TEST ENVIRONMENT FORMATS

In this section the detailed contents of the interface formats are discussed.

Previously, the set-up for testing the RSDIMU versions allowed only minimal control over the generation of the input variables. Consequently, it was not possible to study the effects of gradually changing the values of such variables without repeatedly recompiling the test programs, which would be, of course, senseless. What follows describes the means used to give the experimenter greater control over generation of input values to the versions in the present environment. With such control the effects of each RSDIMU input variable can be studied individually or in combination with other selected input variables in whatever ways might be deemed desirable during the course of the testing.

Each set of input variables, as generated by this control information, is interpreted as a set of *initial conditions*. Given this set of initial conditions, a series of testcases is generated, each with a different acceleration value and set of vehicle frame Euler angles. The number of acceleration values used is given by a control parameter. Within the series of testcases, sensor failure also is simulated as specified by the control information. By keeping the same set of initial conditions for a sequence of acceleration values, the calibration data is kept the same, and the resulting effect is that of performing one calibration of the sensors and then saving that information to be used while performing a series of in flight sensor readings and sets of calculations. In this manner, the capability is achieved for what is hoped to be a reasonable simulation of "flight", with successive sensor readings taken over a period of time.

Sensors are failed on each specified testcase according to their control values (see below). The test drivers in tier-3 have been modified so that for each two consecutive acceleration value and angle sets with the same set of initial conditions, the values for linfailin input to each version are the values for linfailout computed by that version on the previous acceleration and angle set. In this way the various responses of the versions to sensor failure can be studied over a sequence of acceleration values.

The variables that can be controlled are as follows:

linstd	: noise standard deviation for accelerometers
linfailin	: accelerometer failure initial conditions
rawlin	: raw sensor data for acceleration computation
dmode	: display mode
temp	: current temperature on each face
scale0,	
scale1,	
scale2	: linear accelerometer slope coefficients
misalign	: accelerometer misalignment angles
nsigt	: noise tolerance
phii,	
thetai,	
psii	: Euler angles for rotation from the vehicle frame to the instrument frame

Rawlin cannot be controlled directly, as it must be generated by the sensor simulator based on the acceleration, Euler angle, and misalignment angle values. However, whether its value for a given sensor should reflect failure during calibration or failure during flight can be controlled directly,

and so can the value for noise which is used in generating the rawlin value for a sensor which is to be found noisy by the RSDIMU versions. Offraw, the calibration data for the eight accelerometers, can also be indirectly controlled in similar ways, but that control is being left for a later modification of the test control.

For all of the variables above, except for linfailin, misalign, scale0, scale1, scale2, and the sensor failure control information, there are defined three control modes:

- 0 : to indicate that a value should be randomly-generated within a specified range,
- 1 : to indicate that the variable should be varied over a specified range while all other variables except the acceleration values are held constant,
- 2 : to indicate that the variable should be set to a certain specified constant.

For each of these variables the control information is contained on one line of standard input, with the formats as follows:

for mode 0: 0 min max

for mode 1: 1 lowerbound upperbound step

for mode 2: 2 constant

“Min” and “max” specify the range within which the value is to be randomly-generated. “Lowerbound” and “upperbound” specify the range over which the variable is to be varied for mode 1 and “step” specifies the increments by which it is to be varied. “Constant” is the specified value to which the variable is to be set when mode 2 is used. Min, max, lowerbound, upperbound, step, and constant will each be assumed to be of the same type as the RSDIMU input variable which they are being used to control.

For `linfailin` and for the control information regarding which sensors will fail during calibration (equivalent to the output variable `“linnoise”`) the format is slightly different:

for mode 0: 0 lowerbound upperbound

for mode 1: 1 number

for mode 2: 2 bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8

Here the modes are defined as follows:

0 : specifies that a randomly-generated number of the eight values be set to true. This number will be between `“lowerbound”` and `“upperbound”` inclusive. Which of the sensor values are set will be randomly determined.

1 : specifies that `“number”` of the eight values be set to true. Which of the sensor values are set will be randomly determined.

2 : specifies that the eight values be set to the respective constant values, `“bool1”` through `“bool8”`

`“Number”` is assumed to be an integer and `“bool1”` through `“bool8”` will be assumed to be either 0's or 1's, with `“1”` representing true and `“0”` representing false. `“Lowerbound”` and `“upperbound”` will be assumed to be integers between 0 and 8 inclusive.

For `misalign`, `scale0`, `scale1`, and `scale2` the format is as follows:

min max

In this case, since there is only one mode, mode numbers are unnecessary. That mode specifies that each of the 24 `misalign` values (or each of the 8 `scaleX` values) be randomly-generated

between the values "min" and "max", which are assumed to be real numbers and which may be equal.

In addition to the ability to control the values of RSDIMU input variables, it is desirable to have the ability to control which sensors fail during "flight" and during which iteration of sensor reading during the "flight" each sensor fails. To this end the following format is used:

int1 int2 int3 int4 int5 int6 int7 int8

"Int1" through "int8" are integers which represent the sensor reading iteration during which sensors 1 through 8 respectively will fail. A value of 0 for "intX" indicates that sensor X will not fail during this test of the RSDIMU procedures. The non-zero values for "int1" through "int8" must be distinct from one another, as it is assumed in the RSDIMU specifications that at most one sensor will fail on a given sensor reading. The indicated sequence of sensor failures will be simulated once for each set of initial conditions (i.e. for each "flight"). Sensor failure simulation will be accomplished by modifying the generated value for rawlin in such a way that it will appear too noisy to be functional. The modifying value used will be the value for "noise" generated by its control information. Sensors are made to fail not only on the desired sensor reading, but also on all successive readings within a given "flight", so that if a particular RSDIMU version fails to mark that sensor as having failed on that iteration, it may still do so on a subsequent iteration. The value for "intX" should not exceed the value for the number of acceleration values per "flight".

The known acceleration values and the values for phiv, thetav, and psiv are no longer obtained from standard input. Instead, the tier-1 programs write them to a temporary file named by the control parameter "AccelerationFileName" so that they can be used repeatedly (i.e. for each set of "initial conditions").

These control formats have been implemented in such a way that, if two or more variables are being varied over ranges, testcases are generated for all combinations of all the values over which each is being varied, and at the same time conform to any control specifications for other variables.

FILE FORMAT 0 (format for input to tier-1 programs):

CONTROL INFO * | Number of Acceleration Values for each flight
BLOCK | Seed1, Seed2 for random numbers, tiers 1 and 2
| Version Selection Vector (1 element per version, 1 selects
| corresponding version for execution, 0 bypasses)
| AccelerationFileName
| control info for linstd
| control info for linfailin
| control info for number of sensors to fail in calibration
| control info for sensor failure during flight
| control info for noise
| control info for dmode
| control info for temp[1]
| control info for temp[2]
| control info for temp[3]
| control info for temp[4]
| control info for scale0
| control info for scale1
| control info for scale2
| control info for misalign
| control info for nsigt
| control info for phii
| control info for thetai
| control info for psii

FILE FORMAT 1 (format for input to tier-2 programs):

CONTROL INFO BLOCK (see * above)

+ a file containing Number_of_Acceleration_Values lines of

VehicleAccel[x] VehicleAccel[y] VehicleAccel[z] phiv thetav psiv

FILE FORMAT 2 (format for input to tier-3 programs):

CONTROL INFO BLOCK (see * above)

FLIGHT BLOCK (repeated once for each flight)

```
| obase
| offraw[1,1] ... offraw[8,1]
| .
| .
| .
| offraw[1,50] ... offraw[8,50]
| linstd
| linfailin[1] ... linfailin[8] {encoded as integers 0..1}
| dmode
| temp[1] ... temp[4]
| scale0[1] ... scale0[8]
| scale1[1] ... scale1[8]
| scale2[1] ... scale2[8]
| misalign[1,1] ... misalign[1,6]
| misalign[2,1] ... misalign[2,6]
| misalign[3,1] ... misalign[3,6]
| misalign[4,1] ... misalign[4,6]
| nsigt
| phii
| thetai
| psii
| ACCELERATION BLOCK (repeated once per accel value)
|   rawlin[1] ... rawlin[8]
|   normface[1] ... normface[4]
|   phiv thetav psiv
|   KnownBestest.acceleration[x] ... KnownBestest.acceleration[z]
```

FILE FORMAT 3 (format for input to tier-4 programs):

CONTROL INFO BLOCK (see * above)

The Following Repeated For Each Flight:

```
| KnownBestest.accel[x] ... KnownBestest.accel[z]
| The Following Repeated for Each Version Selected For Execution:
|   | linoffset[1] ... linoffset[8]
|   | linnoise[1] ... linnoise[8] {encoded booleans}
|   | linfailout[1] ... linfailout[8] {encoded booleans}
|   | linout[1] ... linout[8]
|   | dismode
|   | disupper[1] ... disupper[3]
|   | dislower[1] ... dislower[3]
|   | bestest.status bestest.acceleration[1] ... bestest.acceleration[3]
|   | chanest[1].status chanest[1].accel[1]...chanest[1].accel[3]
|   | chanest[2].status chanest[2].accel[1]...chanest[2].accel[3]
|   | chanest[3].status chanest[3].accel[1]...chanest[3].accel[3]
|   | chanest[4].status chanest[4].accel[1]...chanest[4].accel[3]
|   | chanface[1] ... chanface[4]
|   | systatus {encoded boolean}
```

SECTION IV

ERROR DETECTION BY SELF TEST EXPERIMENT

In the second experiment, the empirical evaluation of self testing for error detection, we are attempting to determine how well programmers can prepare assertions for the detection of execution-time errors. This study is empirical.

From the set of twenty-seven programs written for the Knight and Leveson experiment [1], eight were chosen for modification. Each of these eight was supplied to three programmers who worked separately to add assertions to the programs. The effort expended by each programmer was one week. The experiment protocol was:

- (1) The program specification was supplied to the programmers and they were given a presentation describing the goals of the experiment, the protocol, and the schedule. Each programmer was also supplied with a copy of the chapter on error detection from the text by Anderson and Lee [2].
- (2) The programmers were required to study the specification and the text on error detection, and then to attempt to develop assertions based purely on knowledge of the specifications.
- (3) When the specification-based assertions were complete, the programmers were supplied with the source text of the program they were to modify. The programs were then modified to include assertions.
- (4) After the assertions had been added, the programmers were supplied with fifteen test cases that executed correctly prior to the addition of the modifications. These test cases should have executed correctly after the addition of the assertions. The programmers were

requested to test the modifications and assertions in any way they chose in addition to the fifteen test cases.

- (5) Finally, the modified programs were subjected to the same set of acceptance tests that had been used in the original experiment [1].

The programmers were asked to keep detailed logs of their effort during the time they were working on the project, and each was required to complete a background technical and educational questionnaire.

Once all three copies of each of the eight programs had been prepared and accepted, the modified programs were tested using the same test driver and test cases that were used in the original experiment. The results of this testing and other analysis are presented in Appendix A.

REFERENCES

- (1) Knight, J.C., N.G. Leveson, and L.D. St.Jean, "A Large-Scale Experiment In N-Version Programming", Digest of Papers FTCS-15: *Fifteenth Annual International Conference on Fault Tolerant Computing*, Ann Arbor, Michigan, June, 1985, pp. 135-139.
- (2) Anderson T., and P.A. Lee, "Fault Tolerance: Principles and Practice", Prentice Hall International, 1981.

APPENDIX A

AN EMPIRICAL STUDY OF SOFTWARE ERROR DETECTION USING SELF-CHECKS

S.D. Cha

N.G. Leveson

T.J. Shimeal

University of California, Irvine

J.C. Knight

University of Virginia

This paper to be presented at the Seventeenth International Symposium on Fault-Tolerant Computing (FTCS17), Pittsburgh, PA.

Abstract

This paper presents the results of an empirical study of error detection using self-checks. A total of twenty-four graduate students in computer science at the University of Virginia and the University of California, Irvine, were hired as programmers. Working independently, each first prepared a set of self-checks using just the specification for an application, and then each modified an existing implementation of the specification. The modified programs were analyzed to classify the various checks that the programmers wrote, and then tested to measure the error-detection performance of the checks.

The goal of this study was not just to obtain quantitative results but to learn more about such checks and how they might best be implemented. This information may result in better methods for formulating checks, making them easier to write and more effective. The analysis of the checks revealed that there are great differences in the ability of individual programmers to design effective checks. We found that some checks that might have been effective failed to detect a fault because they were badly placed, and there were numerous instances of checks signaling non-existent errors. In general, specification-based checks alone were not as effective as combining them with code-based checks.

1. Goals of the Study

Crucial digital systems can fail because of faults in either software or hardware. A great deal of research in hardware design has yielded computer architectures of potentially very high reliability, such as SIFT [Wensley (1978)] and FTMP [Hopkins (1978)]. In addition, distributed systems (incorporating fail-stop processors [Schlichting and Schneider (1983)]) can provide graceful degradation and safe operation even when individual computers fail or are physically damaged.

The state of the art in software development is not as advanced. Current production methods do not yield software with the required reliability for crucial systems, and advanced methods of formal verification [Gries (1981)] and synthesis [Partsch and Steinbruggen (1983)] are not able to deal with software of the required size and complexity. Fault tolerance [Randell (1975)] has been proposed as a technique to allow software to cope with its own faults in a manner reminiscent of the techniques employed in hardware fault tolerance. Many detailed proposals have been made in the literature, but there is little empirical evidence to judge which techniques are most effective or even whether they can be applied successfully to real problems. This study is part of an on-going effort by the authors to collect and examine empirical data on software fault tolerance methods in order to focus future research efforts and to allow decisions to be made about real projects.

Previous studies by the authors have looked at *N*-version programming in terms of independence of failures [Knight and Leveson (1986a)], reliability improvement, and error detection [Knight and Leveson (1986b)]. Other empirical studies of *N*-version programming have been reported [Avizienis and Chen (1977), Gmeiner and Voges (1979), Avizienis and Kelly (1984), Scott *et. al.* (1984), Bishop *et. al.* (1985), and Dunham (1986)]. A study by [Anderson *et. al.* (1985)] showed promise for recovery blocks but concluded that acceptance tests are difficult

to write. Acceptance tests are a subset of the more general run-time assertion or self-check used in exception handling and testing schemes. More information about the use of self-checks to detect software errors might result in better methods for formulating checks, making them easier to write and more effective. Our goal in this study was not merely to provide numerical data but to learn more about such checks and how they might best be implemented.

In order to eliminate as many independent variables from the experiment as possible, it was decided to focus on error detection apart from other issues such as recovery. This also means that the results have implications beyond software fault tolerance alone, for example in the use of embedded assertions to detect software errors during testing [Stucki (1977)]. Furthermore, in some safety-critical systems (e.g., the Boeing 737-300 and the Airbus A310) error detection is the *only* objective. In these systems, software recovery is not attempted and, instead, a non-digital backup system such as an analog or human alternative is immediately given control in the event of a computer system failure. The results of this study may have immediate application in these areas. The next section describes the design of the study. Following this, the results are described and conclusions drawn.

2. Experimental Design

This study uses the programs developed for a previous experiment by [Knight and Leveson (1986a)]. Twenty-seven versions of a program to read radar data and determine whether an interceptor should be launched to shoot down the object (hereafter referred to as the Launch Interceptor Program, or LIP) were prepared from a common specification by graduate students and seniors at the University of Virginia and the University of California, Irvine. Extensive efforts were made to ensure that individual students did not cooperate or exchange information about their program designs during the development phase. The twenty-seven LIP programs

have been analyzed by running one million randomly generated test cases on each program and locating the individual faults that were detected during the testing procedure.

In the present study, 8 students from UCI and 16 students from UVA were employed for a week's time to instrument the programs with self-checking code in an attempt to detect errors in the programs. Eight programs were selected from the 27 and each was randomly assigned to three students (one from UCI and two from UVA). The students were all graduate students in computer science with an average of 2.35 years of graduate study. Professional experience ranged from 0 to 9 years with an average of 1.7 years. None of the participants had prior knowledge of the LIP program nor were they familiar with the results of the previous experiment. There was no significant correlation found between a participant's graduate or industrial experience and their success at writing self-checks.

Participants were provided with a brief explanation of the study along with an introduction to writing self-checks. All also read Chapter 5 on Error Detection from a textbook on fault tolerance [Anderson and Lee (1981)]. The participants were first asked to study the LIP specification and to write checks using only the specification, the training materials, and any additional references the participants desired. When they had submitted their initial checks, they were randomly assigned a program to instrument. The participants were asked to write checks with and without looking at the code in order to determine if there was a difference in effectiveness between self-checks designed by a person working from the requirements alone and those for which the person has access to and information about the program code. On the one hand, the person working only from the requirements might provide more independence by not being influenced by the written code. However, it could also be argued that looking at the code will suggest different and perhaps better self-checks. Because we anticipated that the process of examining the code might result in the participants detecting faults through code-reading alone, participants were asked to report any such detected faults but to still attempt to write a self-check

to detect the fault.

The instrumented versions were subjected to an acceptance test (200 randomly generated test cases) as in the previous experiment. The original versions were known to run correctly on those data, and we wanted to attempt to remove obvious faults introduced by the self-checks. If any false alarms were raised (faults reported that did not actually exist) or if new faults were detected which had been introduced into the program by the instrumentation, the programs were returned to the participants for correction. Along with the instrumented version, participants submitted time sheets, background profile questionnaires, and descriptions of all program faults identified by code reading.

After the instrumented programs had passed the acceptance test, they were executed using the test cases on which they had failed in the previous experiment along with 20,000 new randomly-generated test cases to see if new faults might have been detected. Finally, the self-checks were carefully examined and catalogued as to type of check and effectiveness.

3. Results

The first task of the experiment participants was to read through the program requirements specification and to design self-checks based solely on that specification. These self-checks were found to fall into four groups based on the general strategy of check used:

- [1] *Duplication Checks*: self-checks that duplicate the functionality of the code and compare results. Most, but not all, of the self-checks in this group use algorithms different from the original source code.
- [2] *Structural Checks*: self-checks that verify the proper use of data structures or the proper semantics of code. Examples include a check which verifies that the exit condition of a

loop is true immediately following the loop and a check that verifies that data values have not been improperly overwritten.

- [3] *Reversal Checks*: self-checks that reverse the operation performed by the code and then see if the results are consistent with the input data.
- [4] *Consistency Checks*: self-checks that determine if the results have certain properties. Examples of consistency checks include range checking, arithmetic exception checking, and type checking.

Table 1 shows the classification of the self-checks designed from the specification.[†] The participants labeled 3c and 8b did not provide specification-based self-checks. Note that the largest number of checks written were consistency checks followed by duplication checks. Performance is discussed later, but Tables 4 and 6 show that a total of 33 self-checks were completely or partially effective in detecting errors. Of these 33 effective checks, 4 (or 12%) were formulated by the participants after looking at the requirements specification only. The remaining 88% of the effective checks were designed after the the participants had looked at the code. Although it has been suggested [Anderson and Lee (1981)] that acceptance tests in the recovery block structure must be based on the specification alone, our results indicate that effectiveness of the self-checks can be improved when the specification-based checks are refined and expanded by source code reading and a thorough and systematic instrumentation of the program. It appears that it is very useful for the instrumentor to actually see the code when writing self-checks.

The second task of the participants was to instrument a particular program with self-checks. No limitations were placed on the participants as to how much time could be spent (although they

[†]In order to aid the reader in referring to previously published descriptions of the faults found in the original LIP programs, the programs are referred to in this paper by the numbers previously assigned in the original experiment. A single letter suffix is added (a, b, or c) to distinguish between the three independent instrumentations of the programs.

#	Type of checks used					Total
	Duplication	Structural	Reversal	Consistency	Other*	
3a	1	0	0	10	0	11
3b	1	1	2	10	0	14
6a	2	0	0	0	0	2
6b	0	0	15	9	0	24
6c	0	0	0	14	0	14
8a	20	0	5	0	0	25
8c	15	0	15	16	0	46
12a	16	0	0	0	0	16
12b	0	1	0	26	0	27
12c	1	3	0	0	0	4
14a	16	0	1	0	0	17
14b	21	16	16	36	0	89
14c	2	0	0	4	0	6
20a	0	0	0	4	7	11
20b	15	0	4	34	2	55
20c	8	2	0	5	0	15
23a	17	0	0	0	0	17
23b	0	0	0	27	0	27
23c	0	0	18	15	0	33
25a	15	0	0	0	0	15
25b	0	0	0	8	2	10
25c	0	0	0	5	0	5
Total	149	23	76	218	11	477

Table 1: Specification-Based Self-Checks

were paid only for a 40 hour week which effectively set an upper bound[†]) or how much code could be added. Table 2 describes the change in length in each program during instrumentation. Note that there is a great variation in the amount of code added, ranging from 48 lines to 835 lines. Participants added an average of 37 self-checks, varying from 11 to 97. Despite this variation, there was no correlation between the total number of checks inserted by a participant and the number of those checks that were effective at finding faults. That is, more checks did not

[†]Several reported spending more than 40 hours on the project.

*These self-checks were too vague to be classified

necessarily mean better fault detection.

There was also no statistically significant relationship between the number of hours claimed to have been spent (as reported on the timesheets) by the participants and whether or not they detected any program faults. Table 3 shows the amount of time each participant spent reading the specification and code, developing self-checks based on that reading, implementing the self-checks and debugging the self-checks. The last two columns of the table describe the period of time over which this effort was expended. The "Days Active" column is the number of different dates which appeared on each participant's time-sheet. The "Days Elapsed" column is the number of days between the first and last date on each time-sheet. Three participants (14a, 20a, and 25a) did not submit a time-sheet and are excluded from this table.

Table 4 classifies the program-based self-checks in terms of strategy used and effectiveness. Checks are classified as effective if they correctly report the presence of an error during execution. Two partially effective checks by participant 23a that detect an error most (but not all) of the time are counted as effective. Ineffective checks are those that do not signal an error when one occurs during run-time in the module being checked. False alarms signal an error when no

Version #	Number of Lines				Increase		
	original	a	b	c	a	b	c
3	757	909	1152	805	152	395	48
6	643	859	887	700	216	244	57
8	600	1046	1356	824	446	756	224
12	573	1121	696	806	548	123	233
14	605	905	1342	712	300	737	107
20	533	611	1368	596	78	835	63
23	349	1065	417	544	716	68	195
25	906	1644	1016	1022	738	110	116

Table 2: Lines of Code Added During Instrumentation

#	Hours Spent in Activity					Days	
	Reading	Developing	Coding	Debugging	Total	Active	Elapsed
3a	8	3	7	13	31	6	7
3b	7	17	13.5	5	42.5	4	4
3c	21	7	3	9	40	14	41
6a	2	0.5	9	8	19.5	7	10
6b	6.5	6	13.5	4.5	30.5	4	4
6c	13	6	8	3	30	9	14
8a	14	14	12	12	52	7	9
8b	4.75	5.75	5.75	16.75	33	4	4
8c	8	8	6	15	37	6	7
12a	11	8.5	11.75	11.25	41.5	8	9
12b	7.5	4.5	5.5	3	20.5	5	10
12c	5	3.5	20	13.25	41.75	8	11
14b	4.5	4.25	21.25	9.5	39.5	5	6
14c	5.75	4.75	4	7	21.5	6	6
20b	4	13	9	16	42	5	6
20c	6.5	17.75	5	4	33.25	5	5
23a	4	12	5	10	31	6	7
23b	9.5	5	3.75	8.25	26.5	7	7
23c	3	16	3	10.5	32.5	7	8
25b	7	7	15.5	2.5	32	4	4
25c	8.75	8.5	9	7.25	33.5	6	11

Table 3: Summary of Participant Time-Sheets

error is present. Finally, the effectiveness is classified as unknown if the check does not signal an error and the module being tested is correct.

It can be seen from Table 4 that duplication and consistency checks were about equally effective in detecting faults although more consistency checks were used. For these programs, structural and reversal checks were not effective, but this may have been influenced by the types of faults that were actually in the programs. We examined the ineffective self-checks (checks on code that contained faults but did not detect the faults) in detail. They appear to fail due to one or more of the following reasons:

#	Effectives				Ineffectives				False Alarms				Unknowns				Total
	D	S	R	C	D	S	R	C	D	S	R	C	D	S	R	C	
3a	1	0	0	0	1	2	0	0	0	0	0	0	2	19	0	8	33
3b	0	0	0	0	2	0	0	0	0	0	0	0	11	10	0	13	36
3c	0	0	0	0	0	3	0	0	0	0	0	0	0	11	0	0	14
6a	2	0	0	1	3	2	0	0	0	0	0	0	6	19	1	0	34
6b	0	0	0	0	0	12	0	16	0	1	0	0	0	19	0	34	82
6c	0	0	0	0	0	0	0	9	0	0	0	0	0	0	2	8	19
8a	2	0	0	0	0	0	0	0	0	0	0	0	13	0	0	0	15
8b	0	0	0	0	1	4	0	0	0	1	0	0	6	54	0	2	68
8c	1	0	0	0	1	0	0	0	2	0	0	1	3	1	0	10	19
12a	2	0	0	0	0	0	0	2	0	0	0	0	2	3	0	31	40
12b	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	17	22
12c	1	0	0	8	0	0	0	0	1	0	0	1	0	16	0	8	35
14a	0	0	0	0	0	0	0	5	0	0	0	0	0	1	1	56	63
14b	0	0	0	0	0	0	0	3	0	0	0	0	1	1	23	37	65
14c	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	15	17
20a	0	0	0	0	0	0	0	0	1	0	0	0	4	0	3	3	11
20b	0	0	0	2	2	0	0	1	1	0	0	1	12	0	0	80	99
20c	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	27	29
23a	2*	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	11
23b	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	24	29
23c	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	30	32
25a	7	0	0	2	0	0	0	0	0	0	0	0	0	0	0	31	40
25b	1	0	0	0	0	0	0	0	0	0	0	0	4	0	0	6	11
25c	0	0	0	0	0	5	0	0	0	0	0	0	0	14	0	22	41
Total	19	0	0	14	10	28	0	50	5	2	0	3	73	168	31	462	865

Table 4: Self-Check Classification

- Wrong self-check strategy – the participant used a type of self-check inappropriate to detect the fault present in the code. For example, use of a structural check when the fault was an inadvertent substitution of one variable for another in an expression.
- Wrong check placement – the participant placed the self-check in a location where not all results were checked, and the fault was on a different path.

*These two checks were effective most, but not all, of the time.

- Use of the original faulty code in the self-check – the participant falsely assumed a portion of the code was correct and called that code as part of the self-check.

It should be noted that the placement of the checks may be as crucial as the content. This has important implications for future research in this area and for the use of self-checking in real applications.

It should not be assumed that a false alarm involved a fault in the self-checks. In fact, there were cases where an error message was printed even though both the self-check and the original code were correct. This occurred when the self-check made a calculation using a different algorithm than the original code. Because of the inaccuracies introduced by finite precision arithmetic compounded by the difference in order of operations, the self-check algorithm sometimes produced a result which differed from the original by more than the allowed tolerance. Increasing the tolerance does not necessarily solve this problem in a desirable way. This same problem occurred in our previous experiment and is discussed in detail elsewhere [Brilliant, Knight, and Leveson (1986a)].

Some faults were detected while the participants were reading the code. The numbers in Table 5 refer to the numbering used to identify the individual faults in [Brilliant, Knight, and Leveson (1986b)]. Three faults were reported that actually were not faults; the participant misunderstood the code.

Table 6 summarizes the detected faults by how they were found. 20% of the detected faults were detected by specification-based checks, 40% by code-reading, and 40% by code-based checks. Note that often more than one check detected the same fault in the code-based case, which was not true of the specification-based or code-reading faults.

Version	Fault
3a	3.3
6a	6.1
	6.2
12c	12.1
20b	20.2
20c	20.2
25a	25.1
	25.3

Table 5: Faults Detected Through Code-Reading

Object	Due To			Total
	Spec-based Design	Code Reading	Code-based Design	
Faults Detected	4	8	8	20
Effective Checks	4	8	21	33

Table 6: Fault Detection Classified by Instrumentation Technique

One final way of looking at the results of this study is to consider the number of faults detected and introduced by the participants. Table 7 shows this information. This data makes very clear the difficulty of writing effective self-checks. Of 20 previously known faults in the programs, only 11 were detected (the 14 detected known faults in Table 7 include some multiple detections of the same fault) and only 3 of the 11 detected faults were found by more than one of the three participants instrumenting the same program. It should be noted, however, that the versions used in the experiment are highly reliable (an average 99.9165% success rate on the previous one million case testing), and many of the faults are quite subtle. We could find no particular types of faults that were easier to detect than others. Individual differences in ability appear to be important here.

#	Already Known Faults		Other Faults	Newly Added Faults	
	Present	Detected		Incorrect	NoAnswer
3a	4	1	0	0	0
3b		0	0	0	0
3c		0	0	0	0
6a	3	2	1	0	1
6b		0	0	1	0
6c		0	0	0	0
8a	2	2	0	0	0
8b		0	0	1	0
8c		0	1	3	0
12a	2	1	1	0	0
12b		0	0	0	2
12c		1	1	1	1
14a	2	0	0	0	0
14b		0	0	0	4
14c		0	0	0	0
20a	2	0	0	1	0
20b		1	1	2	0
20c		1	0	0	0
23a	2	2	0	3	1
23b		0	0	0	0
23c		0	0	0	0
25a	3	2	1	0	0
25b		1	0	0	1
25c		0	0	0	0
total	20	14	6	12	10

Table 7: Summary of Error Detection

One rather unusual case occurred. One of the new faults detected by participant 8c was detected quite by accident. There is a previously unknown fault in the program. However, the checking code contains the same fault. An error message is printed because the self-check code uses a different algorithm than the original, and finite precision problems cause the self-check to differ from the original by more than the allowed real-number tolerance. We discovered the new fault while evaluating the error messages printed, but it was entirely by chance. This same thing occurred in modules which did not contain a fault, and in that case the error message was

classified as a false alarm (as discussed above). Our decision was to classify the self-check as effective because it does signal a fault when a fault does exist, but this is a subjective choice.

It is very interesting that the self-checks detected 6 faults not previously detected by comparison of twenty-seven versions of the program with a gold version over a million test cases. After closer examination of the newly discovered faults, we found that one of the reasons they were not uncovered previously is that the strategy of test case selection did not include those test cases that would have revealed the faults. This points out the well-known difficulty in selecting appropriate test cases. The fact that the self-checks uncovered new faults implies that they may have some advantages over voting alone. To understand why, it is instructive to examine an example of one of the previously undetected faults.

Some algorithms are unstable under a few conditions. More specifically, several mathematically valid formulae to compute the area of a triangle are not equally reliable when implemented using finite precision arithmetic. In particular, the use of Heron's formula:

$$area = \sqrt{s * (s - a) * (s - b) * (s - c)}$$

where a , b , and c are the distances between the three points and s is $(a + b + c)/2$, fails in the rare case when all the following conditions are met simultaneously:

- Three points are almost co-linear (but not exactly). s will then be extremely close to one of the distances, say a , so that $(s - a)$ will introduce round-off errors (around 10^{-16} in the hardware employed in this experiment).
- The product of the rest of the terms, $s * (s - b) * (s - c)$, is large enough (approximately 10^4) to make rounding errors significant through multiplication (approximately 10^{-12}).
- The area formed by taking the square root is slightly larger than the real number comparison tolerance (10^{-6} in our example) so that the area is not considered zero.

Other formulas, for example

$$area = \frac{x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1}{2}$$

where x_i and y_i are the coordinates of the three points, did not fail since the potential roundoff errors cannot become “significant” due to the order of operations. 2 of the 6 previously unknown faults detected involved the use of Heron’s formula. Since the source of the unreliability is in the order of computation and inherent in the formula, relaxing the real number comparison tolerance will not prevent this problem. The fault in Heron’s formula was not detected during the previous testing since the voting procedure compared the final result *only*, whereas the self-check verified the validity of the intermediate results as well. For the few cases in which it arose, the faults did not affect the correctness of the final output. However, under different circumstances the final output would have been incorrect.

Although new faults were introduced through the self-checks, this is not very surprising. It is known that changing someone else’s program is difficult and whenever new code is added to a program there is a possibility of introducing faults. All software fault tolerance methods involve adding additional code of one kind or another to the basic application program. The major causes of the new faults were an algorithmic error in a redundant computation, use of an uninitialized variable during instrumentation, logic error, use of Heron’s formula, infinite loops added in instrumentation, out of bounds array reference, etc. The use of uninitialized variables occurred due to incomplete program instrumentation. A participant would declare a temporary variable to hold an intermediate value during the computation, but fail to assign a value on some path through the computation. A rigorous acceptance test may have detected these faults earlier, especially those that cause an abnormal termination of the program.

4. Conclusions

This study was not designed to provide definitive answers to any particular questions, but instead to attempt to determine what the important questions are. This should guide us and others in the design of further experiments, in the evaluation of current proposals, and in the design of new methodologies. Some important questions arise as a result of this study that need to be answered such as:

- [1] There appear to be great differences in individual ability to design effective self-checks. This suggests that more training or experience might be helpful. Our participants had little of either although all were familiar with the use of pre- and post-conditions and assertions to formally verify programs. The data suggests that it might also be interesting to investigate the use of teams to instrument code.
- [2] The programs were instrumented with self-checks in our study by participants who did not write the original code. It would be interesting to compare this with instrumentation by the original programmer. A reasonable argument could be made both ways. The original programmer, who presumably understands the code better, might introduce fewer new faults and might be better able to place the checks. On the other hand, separate instrumentors might be more likely to detect faults since they provide a new view of the problem. More comparative data is needed here.
- [3] Placement of self-checks appeared to cause problems. Some checks that might have been effective failed to detect a fault because they were badly placed. This implies either a need for better decision-making and rules for placing checks or perhaps different software design techniques to make placement easier.

- [4] Specification-based checks alone were not as effective as using them together with code-based checks. This implies that fault tolerance may be enhanced if the alternate blocks in a recovery block scheme, for example, are also augmented with self-checks along with the usual acceptance test. This may also apply to pure voting schemes such as *N*-version programming. A combination of fault-tolerance techniques may be more effective than any one alone. More information is needed on how best to integrate these different proposals.
- [5] The process of writing self-checks is obviously difficult. However, there may be ways to provide help with this process. For example, Leveson and Shimeall (1983) suggest that safety analysis using software fault trees (Leveson and Harvey) can be used to determine the content and the placement of the most important self-checks. Other types of application or program analysis may also be of assistance. Finally, empirical data about common fault types may be important in learning how to instrument code with self-checks.

Many promising research topics, empirical studies, and experiments are suggested by the results of this study that may lead to better procedures for software error detection.

5. Acknowledgements

The authors are pleased to acknowledge the efforts of the experiment participants: David W. Aha, Tom Bair, Jack Beusmans, Bryan Catron, Harry S. Delugach, Siamak Emadi, Lori Fitch, W. Andrew Frye, Joe Gresh, Randy Jones, James R. Kipps, Faith Leifman, Costa Livadas, Jerry Marco, David A. Montuori, John Palesis, Nancy Pomicter, Mary Theresa Roberson, Karen Ruhleder, Brenda Gates Spielman, Yellamraju Venkata Srinivas, Tim Strayer, Gerald Reed Taylor III, and Raymond R. Wagner, Jr.

REFERENCES

- [1] T. Anderson, P.A. Barrett, D.N. Halliwell, and M.R. Moulding, "An Evaluation of Software Fault Tolerance in a Practical System", *Digest of Papers FTCS-15: Fifteenth Annual Symposium on Fault-Tolerant Computing*, pp. 140-145, June 1985.
- [2] T. Anderson, and P.A. Lee, *Fault Tolerance: Principles and Practice* Englewood Cliffs, NJ, Prentice-Hall Intl., 1981.
- [3] A. Avizienis and L. Chen, "On the Implementation of N-version Programming for Software Fault-Tolerance During Execution", *Proceedings of COMPSAC 77* pp. 149-155, November 1977.
- [4] A. Avizienis and J.P.J. Kelly, "Fault Tolerance By Design Diversity: Concepts and Experiments", *IEEE Computer Magazine*, Vol. 17, No. 8, pp. 67-80, August 1984.
- [5] P. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahll, J. Lahti, and S. Yoshimura, "Project on Diverse Software - An Experiment in Software Reliability", *Proceedings of IFAC Workshop SAFECOMP '85*, October 1985.
- [6] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "The Consistent Comparison Problem in N-Version Software", submitted for publication, 1986a.
- [7] S.S. Brilliant, J.C. Knight, and N.G. Leveson, "Analysis of Faults in an N-Version Software Experiment", submitted for publication, 1986b.
- [8] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," *Digest of Papers FTCS-8: Eighth Annual Symposium on Fault Tolerant Computing*, Toulouse, France, pp. 3-9, June 1978.
- [9] J.R. Dunham, "Software Errors in Experimental Systems Having Ultra-Reliability Requirements", *Digest of Papers FTCS-16: Sixteenth Annual Symposium on Fault-Tolerant Computing*, pp 158-164, July 1986
- [10] L. Gmeiner and U. Voges, "Software Diversity in Reactor Protection System: An Experiment", *Proceedings of IFAC Workshop SAFECOMP '79* pp 75-79, 1979
- [11] D. Gries, *The Science Of Programming*, Springer Verlag, 1981.
- [12] A.L. Hopkins, et al., "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor For Aircraft", *Proceedings of the IEEE*, Vol. 66, pp. 1221-1239, October 1978.
- [13] J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming", *IEEE Transaction on Software Engineering*, pp. 96-109, January 1986a.

- [14] J.C. Knight and N.G. Leveson, "An Empirical Study of Failure Probabilities in Multi-Version Software", *Digest of Papers FTCS-16: Sixteenth Annual Symposium on Fault-Tolerant Computing*, pp.165-170, July 1986b.
- [15] N.G. Leveson, and P.R. Harvey, "Analyzing Software Safety", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, pp 569-579, 1983.
- [16] N.G. Leveson, and T.J. Shimeall, "Safety Assertions for Process-Control Systems", *Digest of Papers FTCS-13: Thirteenth Annual Symposium on Fault-Tolerant Computing*, pp 236-240, June 1983.
- [17] H. Partsch and R. Steinbruggen, "Program Transformation Systems", *ACM Computing Surveys*, Vol. 15, No. 3, September 1983.
- [18] B. Randell, "System Structure for Software Fault-Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, pp. 220-232, June 1975.
- [19] R.D. Schlichting and F.B. Schneider, "Fail-Stop Processors: An Approach To Designing Fault-Tolerant Computing Systems", *ACM Transactions On Computer Systems*, Vol. 1, pp. 222-238, August 1983.
- [20] K.R. Scott, J.W. Gault, D.F. McAllister, and J. Wiggs, "Experimental Validation of six Fault Tolerant Software Reliability Models", *Digest of Papers FTCS-14: Fourteenth Annual Symposium on Fault-Tolerant Computing*, pp 102-107, 1984.
- [21] L.G. Stucki, "New Directions in Automated Tools for Improving Software Quality", *Current Trends in Programming Methodology - Volume II: Program Validation*, Prentice Hall, 1977
- [22] J.H. Wensley, et al., "SIFT, The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, Vol. 66, pp. 1240-1254, October 1978.

APPENDIX B

**PRELIMINARY RESULTS OF TESTING THE RSDIMU PROGRAMS
UNDER BENIGN CONDITIONS**

The tests cases that were executed to produce the results described here are termed *benign* because they represent normal operational conditions for the programs. Low levels of residual noise were used and no sensor failures were introduced. Very high reliability would be expected from the programs under these circumstances.

A series of fifteen simulated flights of 1000 acceleration values each were run on 17 of the 20 versions. The three that were not run had too many problems to be of any use (too many runtime failures, no correct responses, corrected version not available). The sensor face temperature and residual noise level were varied systematically. Three different temperature values and five residual noise values were used. The temperatures were 5, 10, and 15 degrees Celsius. In all cases, the residual noise was generated by sampling from a uniform distribution[†]. The five uniform distribution ranges were 0, ± 2 , ± 4 , ± 6 , and ± 8 . Listed below are the values of the other parameters:

Number of Accel Values	1000
Random Number Seeds	28395732 57346274
Noise std dev	fixed at 20
Display mode	fixed at 88
Scale0	fixed at 3.5660e+00
Scale1	fixed at 1.3585e-02
Scale2	fixed at 2.7169e-05
Misalign angles	random betw -1.0000e-01 and 1.0000e-01
Nsigt	fixed at 4
PhiI	random betw 0.0000e+00 and 3.6000e+02
ThetaI	random betw 0.0000e+00 and 3.6000e+02

[†]It is understood that the uniform distribution is unrealistic. This distribution was used merely to allow preliminary evaluation of the programs and detailed debugging of the testing environment.

PsiI

random betw 0.0000e+00 and 3.6000e+02

The results were compared with a gold program (v2.1 provided by NCSU). In order to be considered correct, a version had to agree with the gold on operational status, and if that status was Operational or Analytical, had to agree within a specific tolerance to each of the "X", "Y" and "Z" components of estimated acceleration. Tolerances were relative except when close to zero, when an absolute tolerance (called delta) was used. The versions were compared to the gold using three sets of tolerances:

Tolerance	Inner Limit Around Zero	Delta(Absolute Tolerance)
0.0010	0.0010	0.0001
0.0050	0.0050	0.0005
0.0100	0.0100	0.0010

The individual probabilities of correct responses, the number of times and the number of versions that failed simultaneously, and the number of times that versions produced the same wrong results were computed for each of the three tested tolerances. Fourteen versions performed perfectly under all testing conditions at all tolerances tested. One version performed perfectly under conditions of no noise, and abysmally when any noise was introduced. The other two versions had a very high rate of identical failure. It is interesting to note that these two versions were produced at different universities, yet failed identically so often.

The detailed results of all the tests follow.

RunN1/RunT1:

Noise Val: fixed at 0

Temps: fixed at 5.0000e+00

TOLERANCE = 0.0010

Inner Limit = 0.0010

delta = 0.0001

p(Success) for single versions:

A: 1.0000

B: 1.0000

C: 1.0000

D: 1.0000

E: 1.0000

F: 0.6770

H: 1.0000

J: 1.0000

K: 1.0000

L: 1.0000

M: 1.0000

N: 1.0000

O: 1.0000

P: 0.6810

Q: 1.0000

R: 1.0000

T: 1.0000

Average: 0.9622

Number of simultaneous version failures:

Failures #occur. %

0 670 67.00

1 18 1.80

2 312 31.20

Number of identical version failures:

#versions #occur.

2 289

3 0

TOLERANCE = 0.0050
Inner Limit = 0.0050
delta = 0.0005

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 1.0000
F: 0.9610
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9620
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9955

Number of simultaneous version failures:

Failures	#occur.	%
0	961	96.10
1	1	0.10
2	38	3.80

Number of identical version failures:

#versions	#occur.
2	38
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 1.0000
F: 0.9950
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9940
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9994

Number of simultaneous version failures:

Failures	#occur.	%
0	994	99.40
1	1	0.10
2	5	0.50

Number of identical version failures:

#versions	#occur.
2	5
3	0

RunN1/RunT2:

Noise Val: fixed at 0

Temps: fixed at 1.5000e+01

TOLERANCE = 0.0010

Inner Limit = 0.0010

delta = 0.0001

p(Success) for single versions:

A:	1.0000
B:	1.0000
C:	1.0000
D:	1.0000
E:	1.0000
F:	0.6780
H:	1.0000
J:	1.0000
K:	1.0000
L:	1.0000
M:	1.0000
N:	1.0000
O:	1.0000
P:	0.6730
Q:	1.0000
R:	1.0000
T:	1.0000
Average:	0.9618

Number of simultaneous version failures:

Failures	#occur.	%
----------	---------	---

0	665	66.50
1	21	2.10
2	314	31.40

Number of identical version failures:

#versions	#occur.
-----------	---------

2	286
3	0

TOLERANCE = 0.0050
Inner Limit = 0.0050
delta = 0.0005

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 1.0000
F: 0.9590
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9610
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9953

Number of simultaneous version failures:

Failures	#occur.	%
0	959	95.90
1	2	0.20
2	39	3.90

Number of identical version failures:

#versions	#occur.
2	39
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 1.0000
F: 0.9960
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9960
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9995

Number of simultaneous version failures:

Failures	#occur.	%
0	996	99.60
1	0	0.00
2	4	0.40

Number of identical version failures:

#versions	#occur.
2	4
3	0

RunN1/RunT3:

Noise Val: fixed at 0

Temps: fixed at 2.5000e+01

TOLERANCE = 0.0010

Inner Limit = 0.0010

delta = 0.0001

p(Success) for single versions:

A: 1.0000

B: 1.0000

C: 1.0000

D: 1.0000

E: 1.0000

F: 0.6680

H: 1.0000

J: 1.0000

K: 1.0000

L: 1.0000

M: 1.0000

N: 1.0000

O: 1.0000

P: 0.6700

Q: 1.0000

R: 1.0000

T: 1.0000

Average: 0.9611

Number of simultaneous version failures:

Failures	#occur.	%
----------	---------	---

0	659	65.90
---	-----	-------

1	20	2.00
---	----	------

2	321	32.10
---	-----	-------

Number of identical version failures:

#versions	#occur.
-----------	---------

2	293
---	-----

3	0
---	---

TOLERANCE = 0.0050
Inner Limit = 0.0050
delta = 0.0005

p(Success) for single versions:

A:	1.0000
B:	1.0000
C:	1.0000
D:	1.0000
E:	1.0000
F:	0.9610
H:	1.0000
J:	1.0000
K:	1.0000
L:	1.0000
M:	1.0000
N:	1.0000
O:	1.0000
P:	0.9650
Q:	1.0000
R:	1.0000
T:	1.0000
Average:	0.9956

Number of simultaneous version failures:

Failures	#occur.	%
0	961	96.10
1	4	0.40
2	35	3.50

Number of identical version failures:

#versions	#occur.
2	35
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 1.0000
F: 0.9940
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9950
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9994

Number of simultaneous version failures:

Failures	#occur.	%
0	994	99.40
1	1	0.10
2	5	0.50

Number of identical version failures:

#versions	#occur.
2	5
3	0

RunN2/RunT1

Noise Val: random betw -2 and 2

Temps: fixed at 5.0000e+00

TOLERANCE = 0.0010

Inner Limit = 0.0010

delta = 0.0001

p(Success) for single versions:

A: 1.0000

B: 1.0000

C: 1.0000

D: 1.0000

E: 0.0000

F: 0.6820

H: 1.0000

J: 1.0000

K: 1.0000

L: 1.0000

M: 1.0000

N: 1.0000

O: 1.0000

P: 0.6820

Q: 1.0000

R: 1.0000

T: 1.0000

Average: 0.9038

Number of simultaneous version failures:

Failures	#occur.	%
----------	---------	---

0	0	0.00
---	---	------

1	672	67.20
---	-----	-------

2	20	2.00
---	----	------

3	308	30.80
---	-----	-------

Number of identical version failures:

#versions	#occur.
-----------	---------

2	288
---	-----

3	0
---	---

TOLERANCE = 0.0050
Inner Limit = 0.0050
delta = 0.0005

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0040
F: 0.9620
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9630
Q: 1.0000
R: 1.0000
T: 1.0000

Average: 0.9370

Number of simultaneous version failures:

Failures	#occur.	%
----------	---------	---

0	4	0.40
1	957	95.70
2	3	0.30
3	36	3.60

Number of identical version failures:

#versions	#occur.
-----------	---------

2	36
3	0

TOLERANCE = 0.0100
 Inner Limit = 0.0100
 delta = 0.0010

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.1560
 F: 0.9910
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9910
 Q: 1.0000
 R: 1.0000
 T: 1.0000

Average: 0.9493

Number of simultaneous version failures:

Failures	#occur.	%
0	156	15.60
1	835	83.50
2	0	0.00
3	9	0.90

Number of identical version failures:

#versions	#occur.
2	9
3	0

RunN2/RunT2

Noise Val: random betw -2 and 2
Temps: fixed at 1.5000e+01

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6800
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6760
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9033

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	666	66.60
2	24	2.40
3	310	31.00

Number of identical version failures:

#versions	#occur.
2	288
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0030
 F: 0.9630
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9650
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9371

Number of simultaneous version failures:

Failures	#occur.	%
0	3	0.30
1	958	95.80
2	6	0.60
3	33	3.30

Number of identical version failures:

#versions	#occur.
2	33
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.1410
F: 0.9910
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9910
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9484

Number of simultaneous version failures:

Failures	#occur.	%
0	141	14.10
1	850	85.00
2	0	0.00
3	9	0.90

Number of identical version failures:

#versions	#occur.
2	9
3	0

RunN2/RunT3

Noise Val: random betw -2 and 2
Temps: fixed at 2.5000e+01

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6700
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6680
Q: 1.0000
R: 1.0000
T: 1.0000

Average: 0.9022

Number of simultaneous version failures:

Failures #occur. %

0	0	0.00
1	658	65.80
2	22	2.20
3	320	32.00

Number of identical version failures:

#versions #occur.

2	297
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0010
 F: 0.9620
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9640
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9369

Number of simultaneous version failures:

Failures	#occur.	%
0	1	0.10
1	961	96.10
2	2	0.20
3	36	3.60

Number of identical version failures:

#versions	#occur.
2	36
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.1250
F: 0.9920
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9920
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9476

Number of simultaneous version failures:

Failures	#occur.	%
0	125	12.50
1	867	86.70
2	0	0.00
3	8	0.80

Number of identical version failures:

#versions	#occur.
2	8
3	0

RunN3/RunT1

Noise Val: random betw -4 and 4
Temps: fixed at 5.0000e+00

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6850
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6860
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9042

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	672	67.20
2	27	2.70
3	301	30.10

Number of identical version failures:

#versions	#occur.
2	287
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0000
 F: 0.9570
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9620
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9364

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	957	95.70
2	5	0.50
3	38	3.80

Number of identical version failures:

#versions	#occur.
2	38
3	0

TOLERANCE = 0.0100
 Inner Limit = 0.0100
 delta = 0.0010

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0700
 F: 0.9890
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9900
 Q: 1.0000
 R: 1.0000
 T: 1.0000

Average: 0.9441

Number of simultaneous version failures:

Failures	#occur.	%
0	69	6.90
1	920	92.00
2	2	0.20
3	9	0.90

Number of identical version failures:

#versions	#occur.
2	10
3	0

RunN3/RunT2

Noise Val: random betw -4 and 4

Temps: fixed at 1.5000e+01

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6850
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6810
Q: 1.0000
R: 1.0000
T: 1.0000

Average: 0.9039

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	671	67.10
2	24	2.40
3	305	30.50

Number of identical version failures:

#versions	#occur.
2	289
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0000
 F: 0.9580
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9620
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9365

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	958	95.80
2	4	0.40
3	38	3.80

Number of identical version failures:

#versions	#occur.
2	38
3	0

TOLERANCE = 0.0100
 Inner Limit = 0.0100
 delta = 0.0010

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0610
 F: 0.9920
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9920
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9438

Number of simultaneous version failures:

Failures	#occur.	%
0	60	6.00
1	932	93.20
2	1	0.10
3	7	0.70

Number of identical version failures:

#versions	#occur.
2	8
3	0

RunN3/RunT3

Noise Val: random betw -4 and 4

Temps: fixed at 2.5000e+01

TOLERANCE = 0.0010

Inner Limit = 0.0010

delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6770
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6810
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9034

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	668	66.80
2	22	2.20
3	310	31.00

Number of identical version failures:

#versions	#occur.
2	294
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0000
 F: 0.9650
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9690
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9373

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	965	96.50
2	4	0.40
3	31	3.10

Number of identical version failures:

#versions	#occur.
2	31
3	0

TOLERANCE = 0.0100
 Inner Limit = 0.0100
 delta = 0.0010

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0560
 F: 0.9920
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9920
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9435

Number of simultaneous version failures:

Failures	#occur.	%
0	56	5.60
1	936	93.60
2	0	0.00
3	8	0.80

Number of identical version failures:

#versions	#occur.
2	8
3	0

RunN4/RunT1

Noise Val: random betw -6 and 6

Temps: fixed at 5.0000e+00

TOLERANCE = 0.0010

Inner Limit = 0.0010

delta = 0.0001

p(Success) for single versions:

A: 1.0000

B: 1.0000

C: 1.0000

D: 1.0000

E: 0.0000

F: 0.6870

H: 1.0000

J: 1.0000

K: 1.0000

L: 1.0000

M: 1.0000

N: 1.0000

O: 1.0000

P: 0.6890

Q: 1.0000

R: 1.0000

T: 1.0000

Average: 0.9045

Number of simultaneous version failures:

Failures	#occur.	%
----------	---------	---

0	0	0.00
---	---	------

1	680	68.00
---	-----	-------

2	16	1.60
---	----	------

3	304	30.40
---	-----	-------

Number of identical version failures:

#versions	#occur.
-----------	---------

2	287
---	-----

3	0
---	---

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0000
 F: 0.9680
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9690
 Q: 1.0000
 R: 1.0000
 T: 1.0000

Average: 0.9375

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	968	96.80
2	1	0.10
3	31	3.10

Number of identical version failures:

#versions	#occur.
2	31
3	0

TOLERANCE = 0.0100
 Inner Limit = 0.0100
 delta = 0.0010

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0300
 F: 0.9940
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9940
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9422

Number of simultaneous version failures:

Failures	#occur.	%
0	30	3.00
1	964	96.40
2	0	0.00
3	6	0.60

Number of identical version failures:

#versions	#occur.
2	6
3	0

RunN4/RunT2

Noise Val: random betw -6 and 6

Temps: fixed at 1.5000e+01

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6900
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6910
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9048

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	679	67.90
2	23	2.30
3	298	29.80

Number of identical version failures:

#versions	#occur.
2	280
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0000
 F: 0.9750
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9750
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9382

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	975	97.50
2	0	0.00
3	25	2.50

Number of identical version failures:

#versions	#occur.
2	25
3	0

TOLERANCE = 0.0100
 Inner Limit = 0.0100
 delta = 0.0010

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0280
 F: 0.9950
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9950
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9422

Number of simultaneous version failures:

Failures	#occur.	%
0	28	2.80
1	967	96.70
2	0	0.00
3	5	0.50

Number of identical version failures:

#versions	#occur.
2	5
3	0

RunN4/RunT3

Noise Val: random betw -6 and 6

Temps: fixed at 2.5000e+01

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6890
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6880
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9045

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	677	67.70
2	23	2.30
3	300	30.00

Number of identical version failures:

#versions	#occur.
2	285
3	0

TOLERANCE = 0.0050
Inner Limit = 0.0050
delta = 0.0005

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.9710
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9700
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9377

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	969	96.90
2	3	0.30
3	28	2.80

Number of identical version failures:

#versions	#occur.
2	28
3	0

TOLERANCE = 0.0100
 Inner Limit = 0.0100
 delta = 0.0010

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0190
 F: 0.9940
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9950
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9416

Number of simultaneous version failures:

Failures	#occur.	%
0	19	1.90
1	975	97.50
2	1	0.10
3	5	0.50

Number of identical version failures:

#versions	#occur.
2	5
3	0

RunN5/RunT1

Noise Val: random betw -8 and 8

Temps: fixed at 5.0000e+00

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6900
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6830
Q: 1.0000
R: 1.0000
T: 1.0000

Average: 0.9043

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	676	67.60
2	21	2.10
3	303	30.30

Number of identical version failures:

#versions	#occur.
2	285
3	0

TOLERANCE = 0.0050
Inner Limit = 0.0050
delta = 0.0005

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.9650
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9680
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9372

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	965	96.50
2	3	0.30
3	32	3.20

Number of identical version failures:

#versions	#occur.
2	32
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0260
F: 0.9950
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9950
Q: 1.0000
R: 1.0000
T: 1.0000

Average: 0.9421

Number of simultaneous version failures:

Failures	#occur.	%
0	26	2.60
1	969	96.90
2	0	0.00
3	5	0.50

Number of identical version failures:

#versions	#occur.
2	5
3	0

RunN5/RunT2

Noise Val: random betw -8 and 8

Temps: fixed at 1.5000e+01

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6830
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6840
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9039

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	673	67.30
2	21	2.10
3	306	30.60

Number of identical version failures:

#versions	#occur.
2	287
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0000
 F: 0.9700
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9680
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9375

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	968	96.80
2	2	0.20
3	30	3.00

Number of identical version failures:

#versions	#occur.
2	30
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0230
F: 0.9940
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9950
Q: 1.0000
R: 1.0000
T: 1.0000

Average: 0.9419

Number of simultaneous version failures:

Failures #occur. %

0	23	2.30
1	971	97.10
2	1	0.10
3	5	0.50

Number of identical version failures:

#versions #occur.

2	5
3	0

RunN5/RunT3

Noise Val: random betw -8 and 8

Temps: fixed at 2.5000e+01

TOLERANCE = 0.0010
Inner Limit = 0.0010
delta = 0.0001

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0000
F: 0.6790
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.6760
Q: 1.0000
R: 1.0000
T: 1.0000

Average: 0.9032

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	670	67.00
2	15	1.50
3	315	31.50

Number of identical version failures:

#versions	#occur.
2	292
3	0

TOLERANCE = 0.0050
 Inner Limit = 0.0050
 delta = 0.0005

p(Success) for single versions:

A: 1.0000
 B: 1.0000
 C: 1.0000
 D: 1.0000
 E: 0.0000
 F: 0.9660
 H: 1.0000
 J: 1.0000
 K: 1.0000
 L: 1.0000
 M: 1.0000
 N: 1.0000
 O: 1.0000
 P: 0.9690
 Q: 1.0000
 R: 1.0000
 T: 1.0000
 Average: 0.9374

Number of simultaneous version failures:

Failures	#occur.	%
0	0	0.00
1	966	96.60
2	3	0.30
3	31	3.10

Number of identical version failures:

#versions	#occur.
2	31
3	0

TOLERANCE = 0.0100
Inner Limit = 0.0100
delta = 0.0010

p(Success) for single versions:

A: 1.0000
B: 1.0000
C: 1.0000
D: 1.0000
E: 0.0190
F: 0.9910
H: 1.0000
J: 1.0000
K: 1.0000
L: 1.0000
M: 1.0000
N: 1.0000
O: 1.0000
P: 0.9920
Q: 1.0000
R: 1.0000
T: 1.0000
Average: 0.9413

Number of simultaneous version failures:

Failures	#occur.	%
0	19	1.90
1	972	97.20
2	1	0.10
3	8	0.80

Number of identical version failures:

#versions	#occur.
2	8
3	0

DISTRIBUTION LIST

Copy No.

1 - 3	National Aeronautics and Space Administration Langley Research Center Hampton, Virginia 23665 Attention: Dr. Dave E. Eckhardt, Jr. ISD M/S 478
4 - 5*	NASA Scientific and Technical Information Facility P.O. Box 8757 Baltimore/Washington International Airport Baltimore, Maryland 21240
6 - 7	J. C. Knight, CS
8	A. Catlin, CS
9 - 10	E. H. Pancake, Clark Hall
11	SEAS Publications Files

*1 reproducible copy